CS61B Lecture #23

Today: Backtracking searches, game trees (DSIJ, Section 6.5)
Searching by “Generate and Test”

- We’ve been considering the problem of searching a set of data stored in some kind of data structure: “Is $x \in S$?”

- But suppose we don’t have a set $S$, but know how to recognize what we’re after if we find it: “Is there an $x$ such that $P(x)$?”

- If we know how to enumerate all possible candidates, can use approach of *Generate and Test*: test all possibilities in turn.

- Can sometimes be more clever: avoid trying things that won’t work, for example.

- What happens if the set of possible candidates is infinite?
Backtracking Search

- Backtracking search is one way to enumerate all possibilities.
- Example: *Knight’s Tour.* Find all paths a knight can travel on a chessboard such that it touches every square exactly once and ends up one knight move from where it started.
- In the example below, the numbers indicate position numbers (knight starts at 0).
- Here, knight (N) is stuck; how to handle this?

```
     6     5
  4  7
10  2
 8  3  0
N  9  1
```
/** Append to PATH a sequence of knight moves starting at ROW, COL
* that avoids all squares that have been hit already and
* that ends up one square away from ENDROW, ENDCOL. B[i][j] is
* true iff row i and column j have been hit on PATH so far.
* Returns true if it succeeds, else false (with no change to PATH
* or B). Call initially with PATH containing the starting square, and
* the starting square (only) marked in B. */

boolean findPath(boolean[][] b, int row, int col,
                 int endRow, int endCol, List path) {
    if (path.size() == 64) return isKnightMove(row, col, endRow, endCol);
    for (r, c = all possible moves from (row, col)) {
        if (!b[r][c]) {
            b[r][c] = true; // Mark the square
            path.add(new Move(r, c));
            if (findPath(b, r, c, endRow, endCol, path)) return true;
            b[r][c] = false; // Backtrack out of the move.
            path.remove(path.size()-1);
        }
    }
    return false;
}
Another Kind of Search: Best Move

• Consider the problem of finding the best move in a two-person game.

• One way: assign a heuristic value to each possible move and pick highest (aka static valuation).

• Otherwise, we can use a variety of heuristics. Some examples of static valuations:
  - assign a maximal or minimal value to a won position (depending on side.)
  - number of black pieces – number of white pieces in checkers.
  - (weighted sum of white piece values) – (weighted sum of black pieces in chess), such as queen=9, rook=5, knight=bishop=3, pawn=1.
  - Nearness of pieces to strategic areas (center of board).

• But this is misleading. A move might give us more pieces, but set up a devastating response from the opponent.

• So, for each move, look at opponent’s possible moves, use the best move that results for the opponent as the value.

• But what if you have a great response to opponent’s response?

• How do we organize this sensibly?
Game Trees

- Think of the space of possible continuations of the game as a tree.
- Each node is a position, each edge a move. Suppose numbers at the bottom are the values of those positions to me. Smaller numbers are of more value to my opponent.

- What value can I get if my opponent plays as well as possible? How?
Game Trees, Minimax

- Think of the space of possible continuations of the game as a tree.
- Each node is a position, each edge a move. Numbers are the values we guess for the positions (larger means better for me). Starred nodes would be chosen.

\[ \text{My move} \]
\[ \text{Opponent's move} \]

- I always choose child (next position) with maximum value; opponent chooses minimum value—the \textit{minimax algorithm}. 

\[ \text{My move} \]
\[ \text{Opponent's move} \]
Alpha-Beta Pruning

- We can prune this tree as we search it.

- At the ‘≥ 5’ position, I know that the opponent will not choose to move here (already has a −5 move).

- At the ‘≤ −20’ position, my opponent knows that I will never choose to move here (since I already have a −5 move).

\[
\begin{align*}
\text{My move} & \quad \text{Opponent’s move} \\
\text{My move} & \quad \text{Opponent’s move}
\end{align*}
\]
Cutting off the Search

- If you could traverse game tree to the bottom, you’d be able to force a win (if it’s possible).
- Sometimes possible near the end of a game.
- Unfortunately, game trees tend to be either infinite or impossibly large.
- So, we choose a maximum depth, and use a heuristic static valuation as the value at that depth.
- Or we might use iterative deepening, repeating the search at increasing depths until time is up.
- Much more sophisticated searches are possible, however (take CS188).
**Overall Search Algorithm**

- Depending on whose move it is (maximizing player or minimizing player), we’ll search for a move estimated to be optimal in one direction or the other.
- Search will be exhaustive down to a particular depth in the game tree; below that, we guess values.
- Also pass $\alpha$ and $\beta$ limits:
  - High player does not care about exploring a position further after finding that its value will be larger than a position the minimizing player has already found, because the minimizing player will simply not choose a position with that larger value.
  - Likewise, minimizing player won’t explore a positions whose value is less than what the maximizing player can get ($\alpha$).
- To start, a maximizing player will find a move with the call
  $$\text{maxPlayerValue}(\text{current position}, \text{search depth}, -\infty, +\infty)$$
- minimizing player:
  $$\text{minPlayerValue}(\text{current position}, \text{search depth}, -\infty, +\infty)$$
Sample Tree with Alpha and Beta Values
Some Pseudocode for Searching (Maximizing Player)

/** The estimated minimax value of position POSN, searching up to
 * DEPTH moves ahead, assuming it is the maximizing player’s move.
 * If the value is determined to be <=ALPHA, then the function
 * may return any value <=ALPHA, even if inaccurate. Likewise if the
 * value is >=BETA, it may return any value >=BETA. Assumes ALPHA<BETA. */
int maxPlayerValue(Position posn, int depth, int alpha, int beta)
{
  if (posn \textit{is a final position of the game} || depth == 0)
    return staticGuess(posn);
  int bestSoFar = -\infty;
  for (\textit{each legal move, M, in position} posn) {
    Position next = makeMove(posn, M);
    int response = minPlayerValue(next, depth-1, alpha, beta);
    if (response > bestSoFar) {
      bestSoFar = response;
      alpha = \max(alpha, bestSoFar);
      if (alpha \geq beta)
        return bestSoFar;
    }
  }
  return bestSoFar;
}
Some Pseudocode for Searching (Minimizing Player)

/** The estimated minimax value of position POSN, searching up to *
 * DEPTH moves ahead, assuming it is the minimizing player’s move. */
int minPlayerValue(Position posn, int depth, int alpha, int beta) {
    if (posn is a final position of the game || depth == 0)
        return staticGuess(posn);
    int bestSoFar = +∞;
    for (each legal move, M, in position posn) {
        Position next = makeMove(posn, M);
        int response = maxPlayerValue(next, depth-1, alpha, beta);
        if (response < bestSoFar) {
            bestSoFar = response;
            beta = min(beta, bestSoFar);
            if (alpha >= beta)
                return bestSoFar;
        }
    }
    return bestSoFar;
}